Linear-Time Temporal Logic Control of Discrete Event Systems

Bruno Filipe Araújo Lacerda

Dissertação para a obtenção do grau de Mestre em Matemática e Aplicações

Júri
Presidente: Prof. Cristina S. Sernadas
Orientador: Prof. Pedro A. Lima
Co-Orientador: Prof. F. Miguel Dionísio
Vogal: Prof. Carlos G. Bispo
September 2007
Abstract

A Discrete Event System (DES) is a dynamic system that evolves in accordance with the instantaneous occurrence, at possibly unknown intervals, of physical events. Given a DES, its behavior (the sequence of displayed events) may not satisfy a set of logical performance objectives. The purpose of Supervisory Control is to restrict that behavior in order to achieve those objectives.

Linear-Time Temporal Logic (LTL) is an extension of Propositional Logic which allows reasoning over an infinite sequence of states. We will use this logical formalism as a way to specify our performance objectives for a given DES and build a supervisor that restricts the DES’ behavior to those objectives by construction. Several simulated application examples illustrate the developed method.

Keywords

Discrete Event Systems
Supervisory Control
Languages and Automata
Linear-Time Temporal Logic
Büchi Automata
**Resumo**

Um Sistema de Eventos Discretos (DES) é um sistema dinâmico que evolui de acordo com a ocorrência instantânea, em intervalos de tempo possivelmente desconhecidos, de eventos físicos. Dado um DES, o seu comportamento (a sequência de eventos gerada) pode não satisfazer um conjunto de objectivos lógicos pretendidos. O objectivo do Controlo por Supervisão é restingir esse comportamento de modo a atingir os objectivos.

A Lógica Temporal Linear (LTL) é uma extensão da Lógica Proposicional que permite raciocinar sobre sequências infinitas de estados. Iremos utilizar este formalismo lógico como uma ferramenta para especificar os objectivos de desempenho para um dado DES e construir um supervisor que restringe, por construção, o comportamento do DES para esses objectivos. Vários exemplos simulados de aplicação serão apresentados para ilustrar o método desenvolvido.

**Palavras-Chave**

Sistemas de Eventos Discretos  
Controlo por Supervisão  
Linguagens e Autómatos  
Lógica Temporal Linear  
Autómatos de Büchi
Contents

1 Introduction 3

2 Languages and Automata 5
  2.1 Languages and ω-Languages ........................................... 5
  2.2 Finite State Automata ................................................... 6
  2.3 Operations over Automata ............................................. 11
  2.4 Büchi Automata ........................................................ 12

3 Discrete Event Systems Modeling and Supervision 17
  3.1 Preliminaries .......................................................... 17
  3.2 Modeling Logical DES ............................................... 18
  3.3 Supervisory Control .................................................. 24

4 Linear-Time Temporal Logic 29
  4.1 Definition ............................................................. 29
  4.2 From LTL to Büchi Automata ........................................ 32

5 Supervisor Synthesis 37
  5.1 Defining the LTL-based Supervisor .................................. 37
  5.2 Examples .............................................................. 39

6 Conclusion 45
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>State transition representation of the DFA defined in Example 2.2</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>State transition representation of the NFA defined in Example 2.3</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>DFA $G$ such that $L_m(G) = { s \in E^* : \text{exists } i \in \mathbb{N} \text{ such that } s(i) = s(i+1) = a }$</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>NFA $N$ such that $L_m(N) = { s \in E^* : \text{exists } i \in \mathbb{N} \text{ such that } s(i) = s(i+1) = a }$</td>
<td>10</td>
</tr>
<tr>
<td>2.5</td>
<td>Equivalent DFA to the NFA depicted in Figure 2.2</td>
<td>11</td>
</tr>
<tr>
<td>2.6</td>
<td>Product Composition of automata in Figures 2.1 and 2.2</td>
<td>12</td>
</tr>
<tr>
<td>2.7</td>
<td>Parallel Composition of automata in Figures 2.1 and 2.2</td>
<td>13</td>
</tr>
<tr>
<td>2.8</td>
<td>Büchi Automaton that marks all $\omega$-strings containing $a$ infinitely often</td>
<td>14</td>
</tr>
<tr>
<td>2.9</td>
<td>Büchi Automaton that marks $L = { \sigma \in {a,b}^\omega : \text{exists } i \in \mathbb{N} \text{ such that } \sigma(j) = a \text{ for all } j \geq i }$</td>
<td>15</td>
</tr>
<tr>
<td>2.10</td>
<td>A generalized Büchi automaton, with $X_m = { {x}, {y} }$</td>
<td>16</td>
</tr>
<tr>
<td>2.11</td>
<td>Büchi automaton equivalent to the generalized Büchi automaton depicted in Figure 2.10</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>A simple queueing system</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>FSA model of Transporting Robots</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>FSA model of Transporting Robot $i$</td>
<td>20</td>
</tr>
<tr>
<td>3.4</td>
<td>Illustration of the dining philosophers problem, with 5 philosophers</td>
<td>20</td>
</tr>
<tr>
<td>3.5</td>
<td>FSA model of philosopher $i$</td>
<td>21</td>
</tr>
<tr>
<td>3.6</td>
<td>FSA model of fork $j$</td>
<td>21</td>
</tr>
<tr>
<td>3.7</td>
<td>FSA model of system with two philosophers</td>
<td>22</td>
</tr>
<tr>
<td>3.8</td>
<td>FSA for Robot $R_i$</td>
<td>23</td>
</tr>
<tr>
<td>3.9</td>
<td>The feedback loop of supervisory control</td>
<td>25</td>
</tr>
<tr>
<td>3.10</td>
<td>The feedback loop of modular supervisory control</td>
<td>27</td>
</tr>
</tbody>
</table>
5.1 Büchi automaton marking the $\omega$-strings that satisfy $\varphi_2$ ........................................ 40
5.2 The supervisor $S_2$, obtained by formula $\varphi_2$ ................................................................ 40
5.3 Automaton representation of the controlled system, with the robot alternation implemented ......................................................... 41
Introduction

In recent years there has been a considerable interest in Discrete Event Systems (DES), whose discrete states change in response to the occurrence of events from a predefined event set. Examples of such systems can be found in communication networks, computer programs, operating systems, manufacturing processes and robotics. One of the main fields of study is Supervisory Control, introduced in [9] and further developed in [2], which focuses on the restriction of a DES behavior in order to satisfy a set of performance objectives. This restriction is, in many cases, performed in an ad-hoc manner, but with the continuing growth of these types of systems, a more generalized framework is needed. In this work, we present a framework to restrict a DES behavior specifying the performance objectives with Linear-Time Temporal Logic. Using this theory, we guarantee that the required behavior is achieved by construction. Furthermore, in many cases, the specification of the performances objectives using Linear-Time Temporal Logic is almost immediate, allowing the supervision of more complex systems. A great deal of work has been done recently in a slightly different context: controlling linear systems with LTL specifications ([1] [7] [10]). In this context, a discretization of the linear system is needed before the LTL specification can be enforced, obtaining a discrete system. The system is then refined to a hybrid system. This approach is mainly used to perform robot motion planning (enforcing a robot to go to certain places and avoid certain obstacles). Our approach is a little different as, for example, we will be concerned in, given a team of robots where we assume that each one can perform a number of tasks individually, coordinate their behavior so that they reach a given objective.

The work in divided in four main Chapters: In Chapters 2, 3 and 4, we explain the theory to design Supervisory Control of Discrete Event Systems with specifications given in Linear-Time Temporal Logic and in Chapter 5 we give some operational examples of applications of the presented method.

In Chapter 2 we discuss the theory of languages and automata necessary for the definition of our control framework. We define languages and introduce Finite State Automata as a device ca-
pable of representing a language. We provide the notions of Deterministic and Non-deterministic Finite State Automata and show the equivalence between the two and define the product and parallel composition of Finite State Automata, two operations extremely important in Discrete Event Systems theory. Finally, we approach the theory of $\omega$-languages, which will be relevant in the scope of Linear-Time Temporal Logic and define Büchi automata, referring the equivalence between Büchi automata and generalized Büchi automata and the greater expressing power of non-deterministic Büchi Automata over deterministic Büchi Automata.

In Chapter 3, we introduce the notions of Discrete Event System and Supervisory Control, explaining how one can see a Finite State Automaton as a DES and how Supervisory Control can be also realized by Finite State Automata. We take a look at modular supervisory control, a variant of supervisory control that gives us a gain in complexity.

In Chapter 4, Linear-Time Temporal Logic is defined. We go through its syntax and semantics and give examples of how to represent natural languages statements with Linear-Time Temporal Logic formulas. We also show a method to build a Büchi automaton that accepts exactly the $\omega$-language of the infinite sequences that satisfy a given formula $\varphi$ and prove its correctness. The existence of this method (and mainly others, that present a greater efficiency) is what allows us to perform supervisory control using Linear-Time Temporal Logic formulas.

In Chapter 5, we congregate all the theory defined throughout this work to present our method of supervisory control: Given an uncontrolled DES, we express our performance objectives with Linear-Time Temporal Logic formulas and build the Büchi automata that satisfy the formulas, as explained in Chapter 3. Then, we use these automata to build the supervisors in the traditional way and perform modular control. To allow a better perception of how the supervision is done, we finish by using an implementation of the method we presented to give some examples of controlled systems with Linear-Time Temporal Logic specifications.

Finally, in Chapter 6, we draw some conclusions and describe some ideas for future work.

The main contribution of this work is to compile a set of results, always focusing in one objective: controlling a Discrete Event System using Linear-Time Temporal Logic specifications. Several problems that one may encounter when applying the method are referred and all the theory is illustrated with examples that are developed throughout the work, until the Linear-Time Temporal Logic controlled systems are finally presented.
We start by giving some formal language and automata theory notions that will be used throughout this work.

In Section 1.1, we define languages and \( \omega \)-languages and in Sections 1.2 and 1.4 we introduce Finite State Automata and Büchi Automata as devices capable of representing languages and \( \omega \)-languages respectively, according to well-defined rules. In Section 1.3, we will address some operations over automata that are widely used in the modeling of DES.

One should notice that, since we will use these formalisms to model DES, the nomenclature used in this Chapter is somewhat different from the traditional one used in Computer Science, with the purpose of allowing, later on, a better perception of how the modeling is done.

2.1 Languages and \( \omega \)-Languages

**Definition 2.1 (Language).** A language defined over a set \( E \) is a set of finite sequences \( L \subseteq E^* \) where \( E^* \) is the set of all finite sequences obtained from elements in \( E \).

Languages will be useful in the scope of DES, whose possible behaviors will be represented by a language.

**Definition 2.2 (\( \omega \)-Language).** An \( \omega \)-language defined over a set \( E \) is a set of infinite sequences \( L \subseteq E^\omega \), where \( E^\omega \) is the set of all infinite sequences obtained from elements in \( E \).

\( \omega \)-Languages will be useful in the scope of LTL, where we will be interested in the \( \omega \)-language whose elements satisfy a given formula \( \varphi \) (the meaning of ”satisfying a formula” will be addressed in Chapter 3).

Given a language (\( \omega \)-language) \( L \) over the set \( E \), we call \( E \) its alphabet and the elements of \( L \) its strings (\( \omega \)-strings).
Example 2.1. Let $E = \{a, b, c\}$ be an alphabet. We may define the languages

$$L_1 = \{\epsilon, a, ab\}$$  \hspace{1cm} (2.1)

$$L_2 = \{\text{all possible strings of finite length which start with } a\}$$  \hspace{1cm} (2.2)

$\epsilon$ is used to represent the empty string, i.e., the sequence of 0 elements of $E$.

We can also define the $\omega$-languages

$$L_3 = \{aaaa..., ababab..., abcabcabc...\}$$  \hspace{1cm} (2.3)

$$L_4 = \{\text{all possible strings of infinite length in which } a \text{ appears infinitely often}\}$$  \hspace{1cm} (2.4)

Note that $L_1$ and $L_3$ are finite (both sets have 3 elements) while $L_2$ and $L_4$ are infinite.

Given a string $s = e_0 e_1 e_2 ... e_{n-1} \in E^*$ ($\omega$-string $\sigma = e_0 e_1 e_2 ...$ in $E^\omega$) we denote $s(i)$, $i = 0, ..., n-1$ as the $i+1$-th element of $s$ ($\sigma(i)$, $i \in \mathbb{N}$ as the $i+1$-th element of $\sigma$), $s_i$, $i = 0, ..., n-1$ as the string formed by the first $i$ elements of $s$ ($\sigma_i$, $i \in \mathbb{N}$ as the string formed by the first $i$ elements of $\sigma$) and $|s|$ as the length (number of elements) of $s$. Given $s' = e_0'...e_{m-1}' \in E^*$ ($\sigma' = e_0' e_1' ... \in E^\omega$), $ss' = e_0...e_{n-1}e_0'...e_{m-1}' \in E^*$ ($s\sigma' = e_0...e_{n-1}e_0'... \in E^\omega$) is the concatenation of $s$ and $s'$ (concatenation of $s$ and $\sigma'$).

Remark 2.1. Given a string $s \in E^*$, we can view $s$ as a function, $s : \{0, 1, ..., |s| - 1\} \rightarrow E$, with $s(i)$ giving the element at position $i + 1$ in the string.

Likewise, $\omega$-strings $\sigma \in E^\omega$ can be viewed as functions $\sigma : \mathbb{N} \rightarrow E$.

Next, we define two operations over languages that will be useful later on.

Definition 2.3 (Concatenation of Languages). Let $L_1$ and $L_2$ be two languages over alphabet $E$. The concatenation of $L_1$ and $L_2$ is the language

$$L_1 L_2 = \{s_1 s_2 \in E^* : s_1 \in L_1 \text{ and } s_2 \in L_2\}$$  \hspace{1cm} (2.5)

Definition 2.4 (Prefix Closure of Languages). Let $L$ be a language over alphabet $E$. The prefix closure of $L$ is the language

$$\bar{L} = \{s \in E^* : \exists t \in E^* \text{ such that } st \in L\}$$  \hspace{1cm} (2.6)

A language $L$ such that $L = \bar{L}$ is said to be prefix-closed.

2.2 Finite State Automata

Finite State Automata (FSA) are a simple and compact way of describing a language, according to well-defined rules. In this section we will define Deterministic Finite State Automata and Non-
Deterministic Finite State Automata and address the equivalence between the two formalisms.

Definition 2.5 (Deterministic Finite State Automaton). A Deterministic Finite State Automaton (DFA) is a six-tuple $G = (X, E, f, \Gamma, x_0, X_m)$ where:

- $X$ is the finite set of states
- $E$ is the finite set of events
- $f : X \times E \rightarrow X$ is the (possibly partial) transition function
- $\Gamma : X \rightarrow 2^E$ is the active event function
- $x_0 \in X$ is the initial state
- $X_m \subseteq X$ is the set of marked states

$f(x, e) = y$ means that there is a transition labeled by event $e$ from state $x$ to state $y$. $\Gamma(x)$ is the set of all events $e$ for which $f(x, e)$ is defined. Note that $\Gamma$ is uniquely defined by $f$, it was included in the definition for convenience. We also extend $f$ from domain $X \times E$ to domain $X \times E^*$ in the following recursive manner:

$$f(x, e) = x \quad (2.7)$$

$$f(x, se) = f(f(x, s), e), \; s \in E^*, \; e \in E \quad (2.8)$$

The simplest way to represent a DFA is to consider its state transition diagram.

Example 2.2 (A simple DFA). Let $G = (X, E, f, \Gamma, x_0, X_m)$ where:

- $X = \{x, y\}$
- $E = \{a, b, c\}$
- $f(x, a) = x, \; f(x, b) = y, \; f(y, a) = y, \; f(y, c) = x$
- $\Gamma(x) = \{a, b\}, \; \Gamma(y) = \{a, c\}$
- $x_0 = x$
- $X_m = \{y\}$

In the state transition diagram representation (Figure 2.1), states are represented by circles and transitions by arrows between them, labeled with the correspondent event. The initial state is marked by an arrow and the set of marked states is marked by double circles.

Definition 2.6 (Non-Deterministic Finite State Automaton). A Non-Deterministic Finite State Automaton (NFA) is a six-tuple $N = (X, E, f, \Gamma, X_0, X_m)$ where:

- $X$ is the finite set of states
• $E$ is the finite set of events
• $f : X \times E \to 2^X$ is the (possibly partial) transition function
• $\Gamma : X \to 2^E$ is the active event function
• $X_0 \subseteq X$ is the set of initial states
• $X_m \subseteq X$ is the set of marked states

In an NFA, $y \in f(x, e)$ means that there is a transition labeled by event $e$ from state $x$ to state $y$. Thus, from a given state, there can be several transitions labeled by the same event to different states (hence the non-determinism). Also, the initial state is not unique as in a DFA. Instead, we have a set $X_0$ of possible initial states.

Remark 2.2. It is obvious that a DFA is a special case of a NFA, where $X_0$ is a singleton and, for any $x \in X$ and $e \in E$, if $f(x, e)$ is defined it is also a singleton.

Similarly to the deterministic case, we extend the transition function $f$ from domain $X \times E$ to domain $X \times E^*$:

\[
f(x, e) = \{x\} \quad (2.9)
\]
\[
f(x, se) = \bigcup_{x' \in f(x, s)} f(x', e), \quad s \in E^*, \quad e \in E \quad (2.10)
\]

Example 2.3 (A Simple NFA). Let $N = (X, E, f, \Gamma, X_0, X_m)$ where:

• $X = \{x, y\}$
• $E = \{a, b\}$
• $f(x, a) = \{x\}, \quad f(x, b) = \{y\}, \quad f(y, a) = \{x, y\}$
• $\Gamma(x) = \{a, b\}, \quad \Gamma(y) = \{a\}$
• $X_0 = \{x\}$
• $X_m = \{y\}$
The state transition diagram is represented in Figure 2.2.

We just changed one transition from the DFA in Example 2.2. Now, event $a$ can mean a transition from state $y$ to state $x$ or to itself.

Now, we can define the languages generated and marked by an FSA. We start by defining them for a DFA and then generalize the definitions to an NFA.

**Definition 2.7 (Generated Language by a DFA).** Let $G = (X,E,f,\Gamma,x_0,X_m)$ be a DFA. We define the language generated by $G$ as

$$L(G) = \{ s \in E^* : f(x_0,s) \text{ is defined} \}$$  \hspace{2cm} (2.11)

**Definition 2.8 (Marked Language by a DFA).** Let $G = (X,E,f,\Gamma,x_0,X_m)$ be a DFA. We define the language marked by $G$ as

$$L_m(G) = \{ s \in L(G) : f(x_0,s) \in X_m \}$$  \hspace{2cm} (2.12)

It is obvious that $L_m(G) \subseteq L(G)$ and that $L(G)$ is prefix-closed. The generalization to NFA is straightforward.

**Definition 2.9 (Generated Language by an NFA).** Let $N = (X,E,f,\Gamma,x_0,X_m)$ be an NFA. We define the language generated by $N$ as

$$L(N) = \{ s \in E^* : \exists x_0 \in X_0 \text{ such that } f(x_0,s) \text{ is defined} \}$$  \hspace{2cm} (2.13)

**Definition 2.10 (Marked Language by an NFA).** Let $N = (X,E,f,\Gamma,x_0,X_m)$ be an NFA. We define the language marked by $N$ as

$$L_m(N) = \left\{ s \in L(N) : \left( \bigcup_{x_0 \in X_0} f(x_0,s) \right) \cap X_m \neq \emptyset \right\}$$  \hspace{2cm} (2.14)

**Example 2.4.** Let $E = \{a,b\}$. Consider the language

$$L = \{ s \in E^* : \exists i \in \mathbb{N} \text{ such that } s(i) = s(i+1) = a \}$$  \hspace{2cm} (2.15)

$L$ is the set of all strings that contain the substring 'aa'. In Figure 2.3 we represent a DFA marking $L$ and in Figure 2.4 we represent an NFA marking $L$.  

![Figure 2.2: State transition representation of the NFA defined in Example 2.3](image-url)
Remark 2.3 (Regular Languages). Given an event set $E$, there are languages that cannot be marked by an FSA. One example is the language $L = \{a^n b^n : n \in \mathbb{N}\}$. The languages that can be marked by FSA are designated Regular Languages. These are the languages we will be interested in.

Being a generalization of DFA, one could think that NFA have greater expressing power than DFA, i.e., that there are languages $L$ such that there is an NFA $N$ such that $L_m(N) = L$ but no such DFA $G$ exists. However, this is not true. In fact, given an NFA $N = (X, E, f, \Gamma, X_0, X_m)$, we can build a DFA $G = (X', E, f', \Gamma', x_0', X_m')$ such that $L(G) = L(N)$ and $L_m(G) = L_m(N)$ (we say $G$ is equivalent to $N$):

- $X' = 2^X$
- $f' : 2^X \times E \rightarrow 2^X$ such that
  \[
  f'(x', e) = \bigcup_{x \in x'} f(x, e), \quad x' \in 2^X, \quad e \in E 
  \]  
  (2.16)
- $x_0' = X_0$
- $X_m' = \{x' \in 2^X : x' \cap X_m \neq \emptyset\}$

The idea is to build a DFA that keeps track of all possible states the NFA can be in, i.e., state $\{x_1, ..., x_n\} \in X'$ means that the NFA is in one of the states $x_i$, $i = 1, ..., n$. The proof of the correction of this construction can be found in [5]. Note that the number of states in $G$ is exponential on the number of states in $N$ so, in general, NFA can be used as a more compact way of representing languages.
Example 2.5 (From NFA to DFA). The DFA represented in Figure 2.3 is the result of applying the above construction to the NFA represented in Figure 2.4, where we identify state $x$ as $\{1\}$, state $y$ as $\{1,2\}$ and state $z$ as $\{1,2,3\}$. Note that we omit the states that cannot be reached from state $\{1\}$.

The DFA represented in Figure 2.5 is the result of applying the above construction to the NFA represented in Figure 2.2.

2.3 Operations over Automata

In this section, we define three operations over FSA that will be used in the rest of this work. We already define them for the more general case of NFA.

Definition 2.11 (Accessible Part). Let $G = (X, E, f, \Gamma, X_0, X_m)$ be an FSA. The accessible part of $G$ is the FSA $Ac(G) = (X_{ac}, E_{ac}, f_{ac}, \Gamma_{ac}, X_0, X_{ac,m})$ where

- $X_{ac} = \{x \in X : \exists x_0 \in X_0 \text{ and } s \in E^* \text{ such that } x \in f(x_0, s)\}$
- $X_{ac,m} = X_m \cap X_{ac}$
- $f_{ac} = f|_{X_{ac} \times E \rightarrow 2^X}$
- $\Gamma_{ac} = \Gamma|_{X_{ac} \rightarrow 2^E}$

The accessible part of an FSA is simply its restriction to the states that can be reached from the set of initial states. $f_{ac}$ is the restriction of $f$ to domain $X_{ac} \times E$ and $\Gamma_{ac}$ is the restriction of $\Gamma$ to domain $X_{ac}$. It is clear that $L(G) = L(Ac(G))$ and $L_m(G) = L_m(Ac(G))$. We already applied the accessible part implicitly in Example 2.5.

Definition 2.12 (Product Composition). Let $G_1 = (X_1, E_1, f_1, \Gamma_1, X_{01}, X_{m1})$ and $G_2 = (X_2, E_2, f_2, \Gamma_2, X_{02}, X_{m2})$ be two FSA. The product composition of $G_1$ and $G_2$ is the FSA $G_1 \times G_2 = Ac(X_1 \times X_2, E_1 \cap E_2, f, \Gamma_{1 \times 2}, (X_{01} \times X_{02}), (X_{m1} \times X_{m2}))$ where

$$f((x_1, x_2), e) = \begin{cases} (f_1(x_1) \times f_2(x_2)) & \text{if } e \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

and thus $\Gamma_{1 \times 2}(x_1, x_2) = \Gamma_1(x_1) \cap \Gamma_2(x_2)$
The product composition is also called the completely synchronous composition. In this composition, the transitions of the two FSA must always be synchronized on a common event \( e \in E_1 \cap E_2 \).
This means that an event occurs in \( G_1 \times G_2 \) if and only if it occurs in both FSA. Thus, it is easily verified that \( L(G_1 \times G_2) = L(G_1) \cap L(G_2) \) and \( L_m(G_1 \times G_2) = L_m(G_1) \cap L_m(G_2) \).

**Definition 2.13 (Parallel Composition).** Let \( G_1 = (X_1, E_1, f_1, \Gamma_1, X_{01}, X_{m1}) \) and \( G_2 = (X_2, E_2, f_2, \Gamma_2, X_{02}, X_{m2}) \) be two FSA. The parallel composition of \( G_1 \) and \( G_2 \) is the FSA \( G_1 \parallel G_2 = Ac(X_1 \times X_2, E_1 \cup E_2, f, \Gamma_{1\parallel2}, (X_{01} \times X_{02}), (X_{m1} \times X_{m2})) \) where

\[
f((x_1, x_2), e) = \begin{cases} 
(f_1(x_1) \times f_2(x_2)) & \text{if } e \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\
(f_1(x_1) \times x_2) & \text{if } e \in \Gamma_1(x_1) \setminus E_2 \\
(x_1 \times f_2(x_2)) & \text{if } e \in \Gamma_2(x_2) \setminus E_1 \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

(2.18)

and thus \( \Gamma_{1\parallel2}(x_1, x_2) = [\Gamma(x_1) \cap \Gamma(x_2)] \cup [\Gamma(x_1) \setminus E_2] \cup [\Gamma(x_2) \setminus E_1] \)

The parallel composition is also called the synchronous composition. In this composition, an event in \( E_1 \cap E_2 \) (common event) can only be executed if the two FSA both execute it simultaneously. An event in \( E_1 \setminus E_2 \) (private event of \( G_1 \)) can be executed by \( G_1 \) whenever possible and an event in \( E_2 \setminus E_1 \) (private event of \( G_2 \)) can be executed by \( G_2 \) whenever possible. If \( E_1 = E_2 \), then the parallel composition reduces to the product, since all transitions must be synchronized and if \( E_1 \cap E_2 = \emptyset \), then there are no synchronized transitions and \( G_1 \parallel G_2 \) models the concurrent behavior of \( G_1 \) and \( G_2 \) (in this case we call \( G_1 \parallel G_2 \) the shuffle of \( G_1 \) and \( G_2 \)).

**Example 2.6.** Consider the FSA represented in Figure 2.1 and the one represented in Figure 2.2. The product and parallel compositions of the two automata are depicted in Figures 2.6 and 2.7 respectively.

![Figure 2.6: Product Composition of automata in Figures 2.1 and 2.2](image)

### 2.4 Büchi Automata

Büchi Automata are used to describe a class of \( \omega \)-languages, called \( \omega \)-Regular Languages in a similar way FSA are used to represent Regular Languages. In this section we will define Büchi Automata and will illustrate the superior representation power of Non-Deterministic Büchi Automata over Deterministic Büchi Automata with an example.

**Definition 2.14 (Büchi Automaton).** A Büchi Automaton is a six-tuple \( B = (X, E, f, \Gamma, X_0, X_m) \) where:
Figure 2.7: Parallel Composition of automata in Figures 2.1 and 2.2

- $X$ is the finite set of states
- $E$ is the finite set of events
- $f : X \times E \to X$ (deterministic) or $f : X \times E \to 2^X$ (non-deterministic) is the (possibly partial) transition function
- $\Gamma : X \to 2^E$ is the active event function
- $X_0 \subseteq X$ is the initial state (a singleton for deterministic Büchi Automata)
- $X_m \subseteq X$ is the set of marked states

Note that Büchi automata have the same structure as FSA, so we can extend $f$ to domain $X^* \times E$ in the same way we did for FSA. The characteristic that will set Büchi automata and FSA apart is their semantics, as we define generated and marked languages for FSA while we define generated and marked $\omega$-languages for Büchi automata. Note we will only define these $\omega$-languages to the more general non-deterministic case.

To define the generated and marked $\omega$-languages by a Büchi Automaton, we need to introduce the notion of valid state labeling.

**Definition 2.15 (Valid State Labeling).** Let $B = (X, E, f, \Gamma, X_0, X_m)$ be a Büchi automaton and $\sigma \in E^\omega$ an $\omega$-string. A valid state labeling for $B$ and $\sigma$ is a function $\rho : \mathbb{N} \to X$ such that:

\[
\rho(0) \in X_0 \tag{2.19}
\]
\[
\rho(i + 1) \in f(\rho(i), \sigma(i)), \text{ for all } i \in \mathbb{N} \tag{2.20}
\]

We denote $P(B, \sigma)$ as the set of all possible valid state labelings for $B$ and $\sigma$. 

13
A valid state labeling for $B$ and $\sigma$ is an $\omega$-string over the state set of $B$, where $\rho(i)$ is one of the possible states $B$ can be in (in the deterministic case, the state where $B$ is), while applying its transition function to $\sigma_i$. If, for some $i \in \mathbb{N}$, event $\sigma(i + 1)$ is not active for any of the possible states $B$ can be in, that is,

$$\sigma(i + 1) \not\in \bigcup_{x_0 \in X_0} \left( \bigcup_{x \in f(x_0, \sigma_i)} \Gamma(x) \right)$$

no such function exists.

**Definition 2.16 (Generated $\omega$-Language by a Büchi Automaton).** Let $B = (X, E, f, \Gamma, X_0, X_m)$ be a Büchi automaton. We define the $\omega$-language generated by $B$ as

$$L(B) = \{ \sigma \in E^\omega : P(B, \sigma) \neq \emptyset \}$$

The generated $\omega$-strings by $B$ are the ones for which there exists a valid state labeling.

**Definition 2.17 (Marked $\omega$-Language by a Büchi Automaton).** Let $B = (X, E, f, \Gamma, X_0, X_m)$ be a Büchi automaton. We define the $\omega$-language marked by $B$ as

$$L_m(B) = \{ \sigma \in L(B) : \text{exists } \rho \in P(B, \sigma) \text{ such that } \inf(\rho) \cap X_m \neq \emptyset \}$$

where, for $\chi \in X^\omega$, $\inf(\chi) \subseteq X$ is the set of all $x \in X$ that appear infinite times in $\chi$.

The marked $\omega$-strings by $B$ are the ones generated by "runs" of $B$ that visit at least one of the marked states infinite times.

**Example 2.7.** Let $E = \{a, b\}$ be an event set. Figure 2.8 represents a Büchi Automaton that marks all $\omega$-strings over $E$ containing $a$ infinitely often.

![Figure 2.8: Büchi Automaton that marks all $\omega$-strings containing $a$ infinitely often.](image)

Note that, due to their equal structure, we represent Büchi automata in the same way that we represent FSA. As we said before, what sets a Büchi automaton apart from an FSA is how we define generated and marked language. For example, if we take the state transition diagram in Figure 2.8 as an FSA, it represents the FSA that marks all strings over $E$ that end with event $a$.

**Remark 2.4 (Determinism vs. Non-determinism).** Unlike FSA, Non-deterministic Büchi automata have greater expressive power that deterministic Büchi automata. For example, there is no
deterministic Büchi automaton that marks the \( \omega \)-language \( L = \{ \sigma \in \{a, b\}^\omega : \exists i \in \mathbb{N} \text{ such that } \sigma(j) = a \text{ for all } j \geq i \} \) but the non-deterministic Büchi automaton in Figure 2.9 marks it. We call the languages that can be marked by a non-deterministic Büchi automaton \( \omega \)-regular languages.

![Figure 2.9: Büchi Automaton that marks \( L = \{ \sigma \in \{a, b\}^\omega : \exists i \in \mathbb{N} \text{ such that } \sigma(j) = a \text{ for all } j \geq i \} \)](image)

Remark 2.5. Regarding the operations over FSA defined in the previous section, we can only use the accessible part without having to change the definition. The parallel and product composition versions for Büchi automata will not be defined in this work.

Generalized Büchi Automata are a variant of Büchi automata in which the marked \( \omega \)-language is defined in a slightly different way. First, in a generalized Büchi automaton, we replace the set of marked states \( X_m \subseteq X \) with a set of sets of marked states \( X_m \in 2^X \) and change expression (2.23) in the marked language definition with the following:

\[
L_m(B) = \{ \sigma \in L(B) : \exists \rho \in P(B, \sigma) \text{ such that for all } X \in X_m \inf(\rho) \cap X \neq \emptyset \} \tag{2.24}
\]

A Büchi automaton is the special case of a generalized Büchi Automaton where \( X_m \) is a singleton. Given a generalized Büchi Automaton \( B = (X, E, f, \Gamma, X_0, X_m) \), where \( X_m = \{X_m, 1, \ldots, X_m, k\} \) we can build an equivalent Büchi Automaton \( B' = Ac(X', E, f', \Gamma', X'_0, X'_m) \) where:

- \( X' = X \times \{1, \ldots, k\} \)
- \( f' : (X \times \{1, \ldots, k\}) \times E \to 2^{X \times \{1, \ldots, k\}} \) such that
  \[
  (y, j) \in f'((x, i), e) \text{ if } y \in f(x, e) \text{ and } \begin{cases} j = i & \text{if } x \notin X_{m,i} \\ j = (i \mod k) + 1 & \text{if } x \in X_{m,i} \end{cases} \tag{2.25}
  \]
- \( X_0 = X_0 \times \{1\} \)
- \( X'_m = X_{m,1} \times \{1\} \)

The idea of the construction is, for each state of \( B \), to create \( k \) states labeled in the range \( \{1, \ldots, k\} \). We start with the initial states of \( B \) labeled by 1 and follow \( B' \)'s transition function, keeping the label unchanged except in the case where we go through a state \( x' = (x, j) \in X' \) such that \( x \in X_{m,j} \). In this case we increment the label (resetting to 1 if it is \( k \)). Let's analyze \( B' \)'s generated and marked languages:
• By definition of $f'$, if $y \in f(x,e)$ there is exactly one state $y' \in f'((x,i),e)$ such that $y' = (y,j)$, for all $i,j \in \{1,...,k\}$ ($y'$ will be of the form $(y,i)$ if $x \not\in X_{m,i}$, or of the form $(y,(i \mod k) + 1)$ if $x \in X_{m,i}$). Thus, since $X'_0 = X_0 \times \{1\}$, we can conclude that $L(B') = L(B)$.

• To visit one of the marked states $(x_{m,1},1) \in X_{m}'$ $(x_{m,1} \in X_{m,1})$ infinitely often in $B'$, we have to cycle the labels from 1 to $k$ infinitely often, i.e., we have to visit at least one member of each $X_{m,1},...,X_{m,k}$ infinitely often in $B$. Conversely, if we visit, at least, one member $x_{m,i}$ of each set $X_{m,i}$, $i \in \{1,...,k\}$ infinitely often in $B$ then, after visiting $x_{m,i}$, we will always visit $x_{m,(i \mod k)+1}$ in the future. Thus, we are able to cycle the labels from 1 to $k$ infinitely often in $B'$ and we guarantee that at least $x' = (x_{m,1},1) \in X_{m}'$ is visited infinitely often in $B'$. Therefore, we can conclude that $L_m(B') = L_m(B)$.

**Example 2.8 (From Generalized Büchi to Büchi).** In Figure 2.10 we represent a generalized Büchi automaton, where $X_m = \{\{x\},\{y\}\}$. Figure 2.11 shows the equivalent Büchi automaton, obtained by the above construction.

![Figure 2.10: A generalized Büchi automaton, with $X_m = \{\{x\},\{y\}\}$](image)

![Figure 2.11: Büchi automaton equivalent to the generalized Büchi automaton depicted in Figure 2.10](image)
Discrete Event Systems Modeling and Supervision

In this Chapter we introduce the notion of Discrete Event System (DES) and define a framework for the modeling of these kind of systems. In Sections 2.1 and 2.2, we define DES and explain how an FSA can be seen as a DES and in Section 2.3 we define the theory of Supervisory Control and show how to use an FSA as a supervisor. We leave the examples of supervisory control for Chapter 4, after the synthesis of a supervisor from an LTL formula is addressed.

3.1 Preliminaries

A Discrete Event System (DES) is a dynamic system that evolves in accordance with the abrupt occurrence, at possibly unknown intervals, of physical events. These kind of systems arise in various domains, such as manufacturing, robotics and computer and communication networks.

Definition 3.1 (Discrete Event System). A Discrete Event System is composed of a discrete set $X$ of possible states and a finite set $E = \{e_1, \ldots, e_m\}$ of possible events.

At a given time $t \geq 0$, the DES is always in a certain state $x \in X$, which is all the information needed to characterize the system at that time instant. The state of a DES can only be changed by the occurrence of an event $e \in E$ and these events occur both instantaneously and asynchronously.

The set $X$ is called the state-space of the DES and the set $E$ is called the event-space of the DES. Both these sets must be discrete and $E$ must be finite. We can interpret the state as the task the system if performing at a given moment, such as a robot moving forward, a machine being idle or a computer running a program. The events are interpreted as physical phenomena, such as a robot’s sensor detecting something, a new job arriving to a machine or a program crashing.

Example 3.1 (Queueing System). A queueing system is a system where some “entities” have to wait to use certain resources. These entities are called customers, the resources are called servers and the space where the waiting is done is called queue. For example, if we send several jobs to
a printer, we can think of a queueing system where the jobs are the customers and the printer is the server. The jobs waiting to be printed are said to be in the queue. The queue can have several policies, i.e., there can be several ways of deciding what is the next job to be printed. The most usual is the First-In-First-Out (FIFO) policy, where the next job choosed is the one that is waiting for a longer time. If we think that there can be infinite jobs waiting to be printed, we can consider this queueing system, represented in Figure 3.1, as a DES with $X = \mathbb{N}$ and $E = \{a, d\}$.

The states indicate the number of jobs in the system (the ones in the queue plus the one being printed), the event $a$ stands for the arrival of a job and the event $d$ stands for the departure of a job from the system (after being printed). The occurrence of an event $a$ in state $x \in \mathbb{N}$ changes the state to $x + 1$. The occurrence of an event $d$ in state $x \in \mathbb{N}_1$ changes the state to $x - 1$.

Note that queueing systems can have more than one server, where after exiting a given server, a customer will go to another queue to wait for another server to become available.

**Example 3.2 (Transporting robots)**. Consider two robots, each one holding one end of an iron bar. Their objective is to transport the bar to another place. To simplify, assume that the robots can only move a constant distance forward or stop. This situation can be modeled as a DES with $X = \{\text{Both robots stopped, Robot}_1 \text{ moving and Robot}_2 \text{ stopped, Robot}_1 \text{ stopped and Robot}_2 \text{ moving, Both robots moving}\}$ and $E = \{\text{Move}_1, \text{Move}_2, \text{Stop}_1, \text{Stop}_2\}$.

A sequence of events in this DES can be $((\text{Move}_1, t_1), (\text{Stop}_1, t_2), (\text{Move}_1, t_3), (\text{Move}_2, t_4), (\text{Stop}_1, t_5), (\text{Stop}_2, t_6))$, $t_1 < t_2 < ... < t_6$.

In this example, one of the robots can move forward to a position where it is too far from the other one, making the bar fall. The restriction of their behavior in order to avoid this kind of situation will be adressed in the chapter about Supervisory Control.

### 3.2 Modeling Logical DES

There are three levels of abstraction usually considered in the study of DES:

- **Untimed (or logical) DES models**, where we are just concerned with the order in which the events occurred and ignore the times of occurrence. For example, in a logical model, the sequence in Example 3.2 would be $((\text{Move}_1, \text{Stop}_1, \text{Move}_1, \text{Move}_2, \text{Stop}_1, \text{Stop}_2))$. Two modeling formalisms are mainly used in this level of abstraction: Finite State Automata and Petri Nets.
• **Deterministic Timed DES Models**, used when we want to answer such questions as "How much time does the system spend in a particular state?" or "How soon can a particular state be reached?" and we know the timing of events a priori. Timed Automata and Timed Petri Nets are some of the modeling tools used in this level of abstraction.

• **Stochastic Timed DES Models**, used when the timing of events is not known a priori and must be predicted by suitable statistical assumptions. For this level of abstraction Markov Chains and Generalized semi-Markov Processes are two of the formalisms used to model the DES.

Therefore, FSA can, in fact, be seen as Logical DES. An FSA $G = (X, E, f, \Gamma, X_0, X_m)$ has a set of states, and when an event $e \in E$ occurs in state $x \in X$, the FSA goes to state $x' \in X$, according to its transition function $f$. In this work, we will consider the DES to be deterministic, i.e., when we talk about a DES $G$, $G$ will always be a DFA.

**Example 3.3** (Transporting Robots). Regarding Example 3.2, the DES can be modeled by the FSA shown in Figure 3.2.

![Figure 3.2: FSA model of Transporting Robots](image)

Another way of modeling this system is using parallel composition, which is very useful when our system has several components operating concurrently. It allows us to model each component separately and then get the FSA that models the whole system by applying it. Hence, if we model
each robot separately, we obtain the FSA $G_i, i = 1, 2$, seen in Figure 3.3.

\[ \text{Move}_i \rightarrow S_i \leftarrow \text{Stop}_i \rightarrow M_i \]

Figure 3.3: FSA model of Transporting Robot $i$

It is easy to see that $G_1 \parallel G_2$ is the FSA represented in Figure 3.2.

**Example 3.4 (Dining Philosophers).** The dining philosophers is a traditional problem that illustrates the usage of shared resources by concurrent processes. The problem is stated as a group of $n$ philosophers sitting at a round table with $n$ plates of food and $n$ forks. Figure 3.4 illustrates the problem with 5 philosophers. A philosopher may be "thinking" or he may want to eat. In order to go from the "thinking" state to the "eating" state, the philosopher needs to pick up the fork that is at his left and the fork that is at his right, one at time, in either order. After the philosopher is done eating, he places both forks back on the table (at the same time) and returns to the "thinking" state.

![Figure 3.4: Illustration of the dining philosophers problem, with 5 philosophers](image)

In Figures 3.5 and 3.6, we represent the FSA models of philosopher $i$ and fork $j$ respectively,
\(i, j \in [0, n-1]\), where the events are denoted by \(if_j\) for "philosopher \(i\) picks up fork \(j\)" and \(if\) for "philosopher \(i\) puts both forks down".

A system with \(n\) philosophers is modeled by the parallel composition of the \(n\) FSA representing the philosophers and the \(n\) FSA representing the forks. This models the usage of the resources (the forks), since when a fork is picked up, its state changes to "used" and it cannot be picked up again until it is put back on the table (when that happens its state goes back to "available"). In Figure 3.7, we represent the FSA that models a system with 2 philosophers, which is obtained by the parallel composition of 4 FSA (2 representing the philosophers and 2 representing the forks).

Note that this FSA has two states from which there are no transitions. These states are achieved if both philosophers pick the fork at their right or the fork at their left and then start waiting for the other fork to be available. If the FSA gets to one of those states, it cannot get out of it. This situation is called a deadlock, a particular case of blocking [2].

**Example 3.5** (Robotic Soccer). Consider a team of \(n\) robots playing a soccer game. The objective is to reach a situation in which one of the robots is close enough to the goal to shoot and score. When a robot does not have the ball in its possession, it has two options:
- Move to the ball until it is close enough to take its possession
- Get ready to receive a pass from a teammate

When a robot has the possession of the ball, it can:
- Shoot the ball (if it is close enough to the goal).
- Take the ball to the goal, if there is no opponent blocking its path
- Choose a teammate to pass the ball and, when it is ready to receive, pass it

For simplicity, we assume that, when a robot shoots the ball, the team loses its possession (we do not differentiate the situation where the robot scores from the situation where the robot does not score since the team will lose the ball’s possession in both) and that the opponents do not steal the ball (they are only able to block paths, at which point our robot will try to pass to a teammate). Figure 3.8 depicts a possible FSA $R_i$ model for robot $i$. An FSA model for the whole
team is given by \( T = R_1 \parallel R_2 \parallel \ldots \parallel R_n \). Note that the pass\((i,j)\) event must be synchronized between robot \( i \) (the passing robot) and robot \( j \) (the receiving robot).

![FSA for Robot \( R_i \)](image)

Note that, when we write start_passing\((i,j)\), pass\((i,j)\) and pass\((j,i)\) in a transition, we are representing \( n - 1 \) events, varying \( j \) from 1 to \( n \), \( j \neq i \).
3.3 Supervisory Control

As we have seen in previous examples, sometimes our DES model has some behaviors that are not satisfactory. Let’s assume we have a DES modeled by FSA $G$. $G$ models the "uncotrolled behavior" of the DES and is called the plant. Our objective is to modify the plant’s behavior, i.e., restrict its behavior to an admissible language $L_a \subseteq L(G)$, using control.

To do this, we start by partitioning the event set $E$ in two disjoint subsets

$$E = E_c \cup E_{uc}$$

$E_c$ is the set of controllable events, i.e., the events that can be prevented from happening and $E_{uc}$ is the set of uncontrollable events, i.e., the events that cannot be prevented from happening. This partition is due to the fact that, in general, there are events that make a DES change its state that are not of the "responsibility" of the DES itself.

Example 3.6.
We list the set of controlled and uncontrolled events in previous examples.

- In Example 3.3, we assume that the robots can only move a constant distance forward. Hence, after a robot starts moving, the decision to stop is not its responsibility, it always stops after it moves the predefined distance.

$$E_c = \{\text{Move}_1, \text{Move}_2\} \tag{3.2}$$

$$E_{uc} = \{\text{Stop}_1, \text{Stop}_2\} \tag{3.3}$$

- In Example 3.4, one can consider that a philosopher must always stop eating when he is full. This assumption is acceptable, since a philosopher cannot continue eating for an undefined period of time.

$$E_c = \{i \cdot f_i, i \cdot f_{i+1} \mod n : i = 0, ..., n - 1\} \tag{3.4}$$

$$E_{uc} = \{f : i = 0, ..., n - 1\} \tag{3.5}$$

- In Example 3.5 the events $\text{close} \_\text{to} \_\text{ball}$, $\text{close} \_\text{to} \_\text{goal}$ and $\text{blocked} \_\text{path}$ are caused by changes in the environment around the robots and not by the robots themselves. Therefore, they are considered uncontrollable events. The controllable events correspond to the actions available to each robot.

$$E_c = \{\text{move} \_\text{to} \_\text{ball}(i), \text{get} \_\text{ball}(i), \text{move} \_\text{to} \_\text{goal}(i), \text{kick} \_\text{ball}(i), \text{start} \_\text{passing}(i,j), \text{start} \_\text{receiving}(i), \text{pass}(i,j) : i,j = 1, ..., n, j \neq i\} \tag{3.6}$$

$$E_{uc} = \{\text{close} \_\text{to} \_\text{ball}(i), \text{blocked} \_\text{path}(i), \text{close} \_\text{to} \_\text{goal}(i) : i = 1, ..., n\} \tag{3.7}$$
Next, we introduce the notion of a DES $G = (X, E = E_c \cup E_{uc}, f, \Gamma, x_0, X_m)$ controlled by a supervisor $S$. A supervisor is a function $S : L(G) \rightarrow 2^E$ that, given $s \in L(G)$ outputs the set of events $G$ can execute next (enabled events). We only allow supervisors $S$ such that, when event $e \in E_{uc}$ is active in the plant $G$, it is also enabled by $S$. That is, a supervisor must always allow the plant to execute its uncontrollable events.

**Definition 3.2 (Admissible Supervisor).** Let $G = (X, E = E_c \cup E_{uc}, f, \Gamma, x_0, X_m)$ be a DES and $S : L(G) \rightarrow 2^E$. $S$ is an admissible supervisor for $G$ if, for all $s \in L(G)$

$$E_{uc} \cap \Gamma(f(x_0, s)) \subseteq S(s)$$ (3.8)

**Definition 3.3 (Controlled DES).** Let $G = (X, E = E_c \cup E_{uc}, f, \Gamma, x_0, X_m)$ be a DES and $S : L(G) \rightarrow 2^E$. The controlled DES (CDES) $S/G$ ($S$ controlling $G$) is a DES that constrains $G$ in such a way that, after generating a string $s \in L(G)$, the set of events that $S/G$ can execute next (enabled events) is $S(s) \cap \Gamma(f(x_0, s))$.

The way $S/G$ operates is represented in Figure 3.9 and is as follows: $s$ is the string of all events executed so far by $G$, which is observed by $S$. $S$ uses $s$ to determine what events should be enabled, that is, which events can occur after the generation of $s$.

![Figure 3.9: The feedback loop of supervisory control](image)

**Definition 3.4 (Language generated by a CDES).** Let $S/G$ be a CDES and $e$ one of its events. The language generated by $S/G$, $L(S/G)$, is defined as follows:

- $e \in L(S/G)$
- if $s \in L(G)$ and $se \in L(G)$ and $e \in S(s)$ then $se \in L(S/G)$

From the definition, it is obvious that $L(S/G)$ is prefix-closed.

**Definition 3.5 (Language marked by a CDES).** Let $S/G$ be a CDES. The language marked by $S/G$, $L_m(S/G)$, is

$$L_m(S/G) = L(S/G) \cap L_m(G)$$
Thus, given a plant $G$ and an admissible language $L_a \subseteq L(G)$, we want to find an admissible supervisor $S$ such that $L(S/G) = L_a$ (in this work we will be focused in generated languages and will not be concerned with marked languages). At this point, we cannot guarantee that such supervisor exists. The following theorem gives a necessary and sufficient condition that $L_a$ must oblige to guarantee the existence of $S$.

**Theorem 3.1 (Controllability Theorem).** Let $G = (X, E = E_c \cup E_{uc}, f, \Gamma, x_0, \emptyset)$ and $K \neq \emptyset$. Then there exists an admissible supervisor $S$ such that $L(S/G) = \bar{K}$ if and only if

$$\bar{K} E_{uc} \cap L(G) \subseteq \bar{K}$$

(3.9)

The proof of this theorem can be found in [2].

Condition 3.9 is called the **Controllability Condition** and is defined over the prefix-closure of $K$ because $L(S/G)$ is always prefix-closed (by definition). The controllability condition can be rewritten in a more intuitive way:

$$\text{if } s \in K \text{ and } e \in E_{uc} \text{ and } se \in L(G) \text{ then } se \in \bar{K}$$

(3.10)

The proof is constructive and yields a supervisor $S$ for a language $\bar{K}$ satisfying the controllability condition:

$$S(s) = (E_{uc} \cap \Gamma(f(x_0, s))) \cup \{e \in E_c : se \in \bar{K}\}$$

(3.11)

We will check the satisfaction of the controllability condition in a case by base basis.

In this framework, the supervisor is usually implemented by an FSA $R$, such that $L(R) = L_a$. $R$ is referred to as the **standard realization** of $S$. The most common method to build $R$ is to start by building a simple FSA $H_{\text{spec}}$ that captures the essence of the natural language specification and then combine it with $G$, using either product or parallel composition. We choose parallel composition if the events that appear in $G$ but not in $H_{\text{spec}}$ are irrelevant to the specification that $H_{\text{spec}}$ implements or product composition when, on the other hand, the events that appear in $G$ but not in $H_{\text{spec}}$ should not happen in the admissible behavior $L_a$.

Having the FSA $G = (X_G, E_G, f_G, \Gamma_G, x_{G,0}, X_{G,m})$ and $R = (X_R, E_R, f_R, \Gamma_R, x_{R,0}, X_{R,m})$ that represent the plant and the standard realization of $S$ respectively (note that $E_R \subseteq E_G$), the feedback loop of supervisory control is implemented as follows: Let $G$ be in state $x$ and $R$ be in state $y$ following the execution of string $s \in L(S/G)$. $G$ executes an event $e$ that is currently enabled, i.e., $e \in \Gamma_G(x) \cap \Gamma_R(y)$. $R$ also executes the event, as a passive observer of $G$. Let $x' = f_G(x, e)$ and $y' = f_R(y, e)$ be the new states of $G$ and $R$ respectively, after the execution of $e$. The set of enabled events of $G$ after string $se$ is now given by $\Gamma_G(x') \cap \Gamma_R(y')$. It is common to make $X_{R,m} = X_R$, so that $R \times G$ represents the closed-loop system $S/G$:

- $L(R \times G) = L(R) \cap L(G) = L_a \cap L(G) = L_a = L(S/G)$
- $L_m(R \times G) = L_m(R) \cap L_m(G) = L_a \cap L_m(G) = L(S/G) \cap L_m(G) = L_m(S/G)$
So, from now on, we will refer to a supervisor $S$ and its standard realization $R$ interchangeably.

Next, we address modular supervision, a mean of reducing the complexity of the controlled DES model.

**Definition 3.6 (Modular Supervision).** Let $S_1, ..., S_n$, $n \in \mathbb{N}$ be admissible supervisors for DES $G = (X, E = E_c \cup E_{uc}, f, \Gamma, x_0, X_m)$ and $s \in L(G)$. We define the (admissible) modular supervisor as

$$S_{mod1\ldots n}(s) = S_1(s) \cap S_2(s) \cap \ldots \cap S_n(s) \tag{3.12}$$

It is obvious, by definition 3.2 that $S_{mod1\ldots n}$ is admissible for $G$. In Figure 3.10 we represent modular supervision with 2 supervisors. In modular control, an event is enabled by $S_{mod1\ldots n}$ if and only if it is enabled for all $S_i$, $i = 1, ..., n$.

![Figure 3.10: The feedback loop of modular supervisory control](image)

Let's analyze the gain in efficiency using this method. Given $S_1, ..., S_n$ admissible supervisors for $G$, we could obtain a representation of $S_{mod1\ldots n}$ by building $S_1 \times \ldots \times S_n$. This outputs an FSA with as many as $r_1r_2\ldots r_n$ states, where $r_i \in \mathbb{N}$ is the number of states of $S_i$. On the other hand, if we run each $S_i$ by itself, in parallel, and take the intersection of the active event set of each one (as seen in Figure 3.10 for the $n = 2$ case) we only need a total of $r_1 + \ldots + r_n$ states.

**Remark 3.1 (Multiple Specifications).** When our admissible behavior is composed of multiple specifications, that is, when $L_a = L_{a,1} \cap \ldots \cap L_{a,n}$, where $L_{a,i}$ represents a given specification we want our plant $G$ to satisfy, we will build $n$ supervisors $S_i$, $i = 1, ..., n$ such that $L(S_i/G) = L_{a,i}$ and use modular control to implement a supervisor $S_{mod1\ldots n}$ such that $L(S_{mod1\ldots n}/G) = L_a$.

**Example 3.7 (Transporting Robots).** As we have mentioned, it is possible for one robot to move forward to a position where it is too far from the other, making the bar fall. One way to avoid this is to impose alternation between the robots' motion: one robot moves forward while the other is stopped, holding the bar. Then the other robot moves forward while the one that moved before is stopped, holding the bar, etc. So, we have 4 specifications:
• **Spec 1** - Robot 1 cannot start moving while Robot 2 is moving.

• **Spec 2** - Robot 2 cannot start moving while Robot 1 is moving.

• **Spec 3** - After Robot 1 starts moving, it will only start moving again after Robot 2 has moved.

• **Spec 4** - After Robot 2 starts moving, it will only start moving again after Robot 1 has moved.

**Example 3.8** (Dining Philosophers). As we have seen in Example 3.4, in the dining philosophers a deadlock may occur. One way to solve the problem is by only allowing the philosophers to pick up the forks in a certain order, imposing that the philosophers must always pick up the fork with the smaller index first. Thus, philosophers $i = 0, ..., n-2$ must always pick up the fork at their right (fork $i$) before picking the one at their left (fork $i+1$), and philosopher $n-1$ must always pick up the fork at his left (fork 0) before picking up the one at his right (fork $n-1$). This way, we guarantee that the situations where all philosophers pick up the forks at their right or all the forks at their left never happen (if all philosophers $0, ..., n-2$ pick the fork at their right, the only fork available will be fork $n-1$ and philosopher $n-1$ will not be able to pick it because he must pick fork 0 first. Thus, philosopher $n-2$ will always be able to pick fork $n-1$ and eat, making both forks $n-2$ and $n-1$ available afterwards) and these are the only situations that would cause a deadlock.

**Example 3.9** (Robotic Soccer). Regarding Example 3.5, one may define the following specifications, which are useful to improve the team’s performance in a soccer game for each Robot $i$:

• **Spec 1, $i$** - If another teammate goes to the ball, robot $i$ will not go to the ball until it is kicked by some robot in the team.

• **Spec 2, $i$** - Robot $i$ will not get ready to receive a pass, unless one of its teammates decides to pass it the ball and, in this case, it will be ready to receive the pass as soon as possible.

Spec 1, $i$ guarantees that only one robot moves to the ball at a time and that, when the team has the ball, no robot moves to it and Spec 2, $i$ guarantees that no robot will be ready to receive a pass when none of its teammates wants it to receive it and that when a robot wants to pass the ball, another one will get ready to receive it as soon as possible.
In this Chapter we introduce Linear-Time Temporal Logic (LTL) and prove that, for every LTL formula $\varphi$ there exists a Büchi automaton $B_\varphi$ that marks exactly the $\omega$-strings that satisfy $\varphi$. In Section 3.1 we define the syntax and semantics of LTL and in Section 3.2 we show how to build the Büchi automaton $B_\varphi$ and while proving the construction’s correctness.

4.1 Definition

LTL is an extension of Propositional Logic which allows reasoning over an infinite sequence of states. LTL is widely used for verification of properties of several concurrent systems (for example, safety and liveness), especially software systems. In the following, $\Pi$ is a set of propositional symbols.

Definition 4.1 (Syntax). The set $L_{LTL}(\Pi)$ of LTL formulas over $\Pi$ is defined inductively as follows:

- $true, false \in L_{LTL}(\Pi)$
- If $p \in \Pi$ then $p \in L_{LTL}(\Pi)$
- If $\varphi, \psi \in L_{LTL}(\Pi)$ then $\lnot \varphi, (\varphi \lor \psi), (\varphi \land \psi) \in L_{LTL}(\Pi)$
- If $\varphi \in L_{LTL}(\Pi)$ then $(X \varphi) \in L_{LTL}(\Pi)$
- If $\varphi, \psi \in L_{LTL}(\Pi)$ then $(\varphi U \psi) \in L_{LTL}(\Pi)$
- If $\varphi, \psi \in L_{LTL}(\Pi)$ then $(\varphi R \psi) \in L_{LTL}(\Pi)$

In Definitions 4.2 and 4.3, we define the LTL semantics.

Definition 4.2 (Local Satisfaction). Let $\sigma : \mathbb{N} \to 2^\Pi$, $t \in \mathbb{N}, p \in \Pi$ and $\varphi, \psi \in L_{LTL}(\Pi)$. The notion of satisfaction ($\models$) is defined as follows:
• $\sigma(t) \models \text{true}$ and $\sigma(t) \not\models \text{false}$

• $\sigma(t) \models p$ if and only if $p \in \sigma(t)$

• $\sigma(t) \models (\neg \varphi)$ if and only if $\sigma(t) \not\models \varphi$

• $\sigma(t) \models (\varphi \lor \psi)$ if and only if $\sigma(t) \models \varphi$ or $\sigma(t) \models \psi$

• $\sigma(t) \models (\varphi \land \psi)$ if and only if $\sigma(t) \models \varphi$ and $\sigma(t) \models \psi$

• $\sigma(t) \models (X \varphi)$ if and only if $\sigma(t + 1) \models \varphi$

• $\sigma(t) \models (\varphi U \psi)$ if and only if exists $t' \geq t$ such that $\sigma(t') \models \psi$ and for all $t'' \in [t, t']$ $\sigma(t'') \models \varphi$

• $\sigma(t) \models (\varphi R \psi)$ if and only if for all $t' \geq t$ such that $\sigma(t') \not\models \psi$ exists $t'' \in [t, t']$ such that $\sigma(t'') \models \varphi$

**Definition 4.3** (Global Satisfaction). Let $\sigma : \mathbb{N} \to 2^\Pi$ and $\varphi \in L_{\text{LTL}}(\Pi)$. The notion of global satisfaction is defined as follows:

• $\sigma \models \varphi$ if and only if $\sigma(0) \models \varphi$

As we have seen in Remark 2.1, we can say that LTL formulas are interpreted over $\omega$-strings $\sigma$ over the alphabet $2^\Pi$. Each $\sigma(t), t \in \mathbb{N}$ is called a state. In Definition 4.2, we give the conditions for the satisfaction of an LTL formula $\varphi$ by a state $\sigma(t), t \in \mathbb{N}$ and in Definition 4.3, we give the condition for the satisfaction of an LTL formula $\varphi$ by an $\omega$-string $\sigma$: its first state ($\sigma(0)$) must satisfy $\varphi$.

Now, we give a brief explanation of each operator defined:

• The $X$ operator is read "next", meaning that the formula it precedes will be true in the next state

• The operator $U$ is read "until", meaning that its first argument will be true until its second argument becomes true (and the second argument must become true in some state, i.e., an $\omega$-string where $\varphi$ is always satisfied but $\psi$ is never satisfied does not satisfy $\varphi U \psi$)

• The operator $R$, which is the dual of $U$, is read "releases", meaning that its second argument must always be true until its first argument becomes true (in this case, an $\omega$-string where $\psi$ is always satisfied satisfies $\varphi R \psi$, because the definition does not require the existence of $t'$)

There are two other commonly used temporal operators, $F$ and $G$, usually defined by abbreviation.

**Definition 4.4** (Abbreviations). Let $p \in \Pi$ and $\varphi, \psi \in L_{\text{LTL}}(\Pi)$. We define the following abbreviations:

• $(\varphi \Rightarrow \psi) \equiv_{\text{abv}} ((\neg \varphi) \lor \psi)$

• $(\varphi \Leftrightarrow \psi) \equiv_{\text{abv}} ((\varphi \Rightarrow \psi) \land (\psi \Rightarrow \varphi))$

30
\[ F\varphi \equiv_{abv} (\text{true} U \varphi) \]

\[ G\varphi \equiv_{abv} (\text{false} R \varphi) \]

- The \( F \) operator is read "eventually", meaning that the formula it precedes will be true in a future state.
- The \( G \) operator is read "always", meaning the formula it precedes will be true in all future states.

**Example 4.1** (Interpreting LTL formulas in natural language). To gain some intuition about what can be expressed in LTL, we will "translate" some formulas to natural language.

- \((G(F\varphi))\) means that \( \varphi \) must be satisfied infinitely often (\( \varphi \) will always be eventually true).
- \((F(G\varphi))\) means that from some state on, \( \varphi \) will be always satisfied (\( \varphi \) will eventually become always true).
- \((\varphi U(G\psi))\) means that \( \varphi \) must always be true until a state where \( \psi \) is always true in future states.
- \((G(\varphi \Rightarrow (X\psi)))\) means that for all states that satisfy \( \varphi \), the next state must satisfy \( \psi \).

**Remark 4.1** (Normal Negated Form). We say a formula is in the normal negated form if the negation (\( \neg \)) operator is only applied to propositional symbols \( p \in \Pi \). Using the following identities, which are direct consequences of the LTL semantics, we can always rewrite an LTL formula in the normal negated form:

\[
\neg \text{false} = \text{true} \quad (4.1)
\]
\[
\neg \text{true} = \text{false} \quad (4.2)
\]
\[
\neg(\neg \varphi) = \varphi \quad (4.3)
\]
\[
\neg(\varphi \lor \psi) = (\neg \varphi) \land (\neg \psi) \quad (4.4)
\]
\[
\neg(\varphi \land \psi) = (\neg \varphi) \lor (\neg \psi) \quad (4.5)
\]
\[
\neg(\text{X}\varphi) = (\text{X}(\neg \varphi)) \quad (4.6)
\]
\[
\neg(\varphi U \psi) = ((\neg \varphi) R (\neg \psi)) \quad (4.7)
\]
\[
\neg(\varphi R \psi) = ((\neg \varphi) U (\neg \psi)) \quad (4.8)
\]

From now on, without losing generality, we will assume that all formulas are in the normal negated form.

**Example 4.2** (Rewriting to normal negated form). Let \( \Pi = \{p, q\} \) and \( \varphi = (\neg(pU(Gq))) \). Let’s rewrite \( \varphi \) in the normal negated form:

\[
\varphi = (\neg(pU(Gq))) \equiv_{abv} (\neg(pU(\text{false} R q))) = ((\neg p) R (\neg (\text{false} R q))) =
\]
\[
= ((\neg p) R ((\neg \text{false} U (\neg q))) = ((\neg p) R (\text{true} U (\neg q))) \quad (4.9)
\]
4.2 From LTL to Büchi Automata

In this section we will develop a procedure to build a Büchi automaton that marks the \( \omega \)-language of exactly the strings that satisfy a given formula \( \varphi \). All Definitions and proofs in this Section are an adaptation of [11]. Keep in mind we assume \( \varphi \) to be in the normal negated form. We start by defining some notions that will be necessary for the construction.

**Definition 4.5 (Closure of a Formula).** Let \( \varphi \in L_{LT}(\Pi) \). We define the closure of \( \varphi \) inductively as follows:

- \( \varphi \in \text{cl}(\varphi) \)
- if \( \varphi_1 \land \varphi_2 \in \text{cl}(\varphi) \) then \( \varphi_1, \varphi_2 \in \text{cl}(\varphi) \)
- if \( \varphi_1 \lor \varphi_2 \in \text{cl}(\varphi) \) then \( \varphi_1, \varphi_2 \in \text{cl}(\varphi) \)
- if \( X \varphi_1 \in \text{cl}(\varphi) \) then \( \varphi_1 \in \text{cl}(\varphi) \)
- if \( \varphi_1 U \varphi_2 \in \text{cl}(\varphi) \) then \( \varphi_1, \varphi_2 \in \text{cl}(\varphi) \)
- if \( \varphi_1 R \varphi_2 \in \text{cl}(\varphi) \) then \( \varphi_1, \varphi_2 \in \text{cl}(\varphi) \)

**Example 4.3.**

\[
\text{cl}(G(p \Rightarrow q)) = \text{cl}(\text{falseR}(p \Rightarrow q)) = \text{cl}(\text{falseR}((\neg p) \lor q)) = \{((\text{falseR}((\neg p) \lor q)), false, ((\neg p) \lor q), (\neg p), q) \}
\] (4.10)

We now establish a class of functions \( \tau : \mathbb{N} \rightarrow 2^{\text{cl}(\varphi)} \) as the functions that satisfy a set of rules. These rules are defined to mirror the semantics of LTL.

**Definition 4.6 (Valid Closure Labeling).** Let \( \varphi \in L_{LT}(\Pi) \) and \( \sigma : \mathbb{N} \rightarrow 2^{\Pi} \). A valid closure labeling of \( \sigma \) is a function \( \tau : \mathbb{N} \rightarrow 2^{\text{cl}(\varphi)} \) such that the following rules are satisfied:

1. \( \text{false} \notin \tau(i) \)
2. for all \( p \in \Pi \), if \( p \in \tau(i) \) then \( p \in \sigma(i) \) and if \( (\neg p) \in \tau(i) \) then \( p \notin \sigma(i) \)
3. if \( (\varphi_1 \land \varphi_2) \in \tau(i) \) then \( \varphi_1 \in \tau(i) \) and \( \varphi_2 \in \tau(i) \)
4. if \( (\varphi_1 \lor \varphi_2) \in \tau(i) \) then \( \varphi_1 \in \tau(i) \) or \( \varphi_2 \in \tau(i) \)
5. if \( (X \varphi_1) \in \tau(i) \) then \( \varphi_1 \in \tau(i+1) \)
6. if \( (\varphi_1 U \varphi_2) \in \tau(i) \) then either \( (\varphi_2 \in \tau(i)) \) or \( (\varphi_1 \in \tau(i) \) and \( (\varphi_1 U \varphi_2) \in \tau(i+1)) \)
7. if \( (\varphi_1 R \varphi_2) \in \tau(i) \) then \( \varphi_2 \in \tau(i) \) and either \( \varphi_1 \in \tau(i) \) or \( (\varphi_1 R \varphi_2) \in \tau(i+1) \)
8. if \( (\varphi_1 U \varphi_2) \in \tau(i) \) then there exists \( j \geq i \) such that \( \varphi_2 \in \tau(j) \)
Rules 1 to 5 are an immediate consequence of the definition of satisfaction. Rules 6 and 7 concern the operators $U$ and $R$ respectively and are obtained from the following equivalences, that can be easily proven from the LTL semantics for all $\sigma : \mathbb{N} \rightarrow 2^\Pi$ and $t \in \mathbb{N}$:

\[
\sigma(t) \models (\varphi_1 U \varphi_2) \text{ if and only if } \sigma(t) \models (\varphi_2 \lor (\varphi_1 \land (X(\varphi_1 U \varphi_2)))) \tag{4.11}
\]

\[
\sigma(t) \models (\varphi_1 R \varphi_2) \text{ if and only if } \sigma(t) \models (\varphi_2 \land (\varphi_1 \lor (X(\varphi_1 R \varphi_2)))) \tag{4.12}
\]

This way, we can define rules that, for each $i \in \mathbb{N}$, only take into account $\tau(i)$ and $\tau(i+1)$.

Rule 8 also concerns the $U$ operator, and must be introduced, because, as we have referred before, the $U$ operator is only satisfied if there is a state such that its second argument is satisfied and rule 6 does not enforce that.

The following lemma states that the valid closure labelings $\tau : \mathbb{N} \rightarrow 2^{cl(\varphi)}$ of $\sigma$ are functions that, for each $i \in \mathbb{N}$, only contain formulas that are satisfied by $\sigma$ in state $i$. Hence the name valid closure labeling: they label each state of $\sigma$ with a set of formulas contained in $cl(\varphi)$ that are satisfied by $\sigma$ in that state.

**Lemma 4.1.** Let $\varphi \in L_{\text{LTL}}(\Pi)$ and $\sigma : \mathbb{N} \rightarrow 2^\Pi$. If $\tau : \mathbb{N} \rightarrow 2^{cl(\varphi)}$ is a valid closure labeling then

\[
\text{if } \psi \in \tau(i) \text{ then } \sigma(i) \models \psi \tag{4.13}
\]

**Proof.** The proof of this lemma follows by structural induction on the formulas $\psi \in cl(\varphi)$.

**Basis**

- $\psi = \text{true}$
  Immediate, by definition of satisfaction.

- $\psi = \text{false}$
  By rule 1, $\text{false} \not\in \tau(i)$. Therefore the condition is vacuously true.

- $\psi = p$, with $p \in \Pi$
  If $p \in \tau(i)$, then, by rule 2, $p \in \sigma(i)$ and, by definition of satisfaction, $\sigma(i) \models p$.

**Step**

- $\psi = (\neg p)$, with $p \in \Pi$
  If $(\neg p) \in \tau(i)$, then, by rule 2, $p \not\in \sigma(i)$ and, by definition of satisfaction, $\sigma(i) \not\models p$, thus $\sigma(i) \models (\neg p)$.

- $\psi = (\varphi_1 \land \varphi_2)$
  If $(\varphi_1 \land \varphi_2) \in \tau(i)$ then, by rule 3, $\varphi_1 \in \tau(i)$ and $\varphi_2 \in \tau(i)$ and, by inductive hypothesis, $\sigma(i) \models \varphi_1$ and $\sigma(i) \models \varphi_2$. Hence, by definition of satisfaction, $\sigma(i) \models (\varphi_1 \land \varphi_2)$.
• \( \psi = (\varphi_1 \lor \varphi_2) \)
  Analogous to previous case, by rule 4.

• \( \psi = (X\varphi_1) \)
  Analogous to previous case, by rule 5.

• \( \psi = (\varphi_1 U \varphi_2) \)
  If \( (\varphi_1 U \varphi_2) \in \tau(i) \) then, by rule 8, exists \( j \geq i \) such that \( \varphi_2 \in \tau(j) \) and, by inductive hypothesis, \( \sigma(j) \models \varphi_2 \). Take the smallest such \( j \) and let \( k \in [i, j[. \) We have that, for all \( k' \in [i, k[, \varphi_2 \notin \tau(k') \). Since \( (\varphi_1 U \varphi_2) \in \tau(i) \), by rule 6, \( (\varphi_1 U \varphi_2) \in \tau(k) \) and \( \varphi_1 \in \tau(k) \).

Hence, by inductive hypothesis, \( \sigma(k) \models \varphi_1 \). Since we chose \( k \in [i, j[ \) arbitrarily, by definition of satisfaction we have \( (\varphi_1 U \varphi_2) \models \sigma(i) \).

• \( \psi = (\varphi_1 R \varphi_2) \)
  If \( (\varphi_1 R \varphi_2) \in \tau(i) \) then, by rule 7, \( \varphi_2 \in \tau(i) \) and, by inductive hypothesis, \( \sigma(i) \models \varphi_2 \). Let \( j > i \) such that \( \varphi_2 \notin \tau(j) \).

Hence, by rule 7, \( (\varphi_1 R \varphi_2) \notin \tau(j) \) and, by inductive hypothesis, \( \sigma(j) \not\models \varphi_2 \). We have to find \( k \in [i, j[ \) such that \( \sigma(k) \models \varphi_1 \). Let \( k \) be the smallest natural \( \geq i \) such that \( (\varphi_1 R \varphi_2) \notin \tau(k+1) \) (we already know that \( k \in [i, j[ \)). We have that \( (\varphi_1 R \varphi_2) \in \tau(k) \) and, by rule 7, \( \varphi_1 \in \tau(k) \).

Thus, by inductive hypothesis, \( \sigma(k) \models \varphi_1 \) and we can conclude, by definition of satisfaction, that \( \sigma(i) \models (\varphi_1 R \varphi_2) \).

If no such \( j \) exists, by inductive hypothesis, \( \sigma(k) \models \varphi_2 \) for all \( k \geq i \), hence \( \sigma(i) \models (\varphi_1 R \varphi_2) \) (the satisfaction condition for the \( R \) operator is satisfied vacuously). \( \blacksquare \)

Now, we establish that when an \( \omega \)-string satisfies a formula, a valid closure labeling exists.

**Lemma 4.2.** Let \( \varphi \in L_{\mathit{LTL}}(\Pi) \) and \( \sigma : \mathbb{N} \rightarrow 2^\Pi \). If \( \sigma \models \varphi \), then there exists a valid closure labeling \( \tau : \mathbb{N} \rightarrow 2^{cl(\varphi)} \) of \( \sigma \) such that \( \varphi \in \tau(0) \).

**Proof.** Let \( \tau \) be defined as

\[
\psi \in \tau(i) \text{ if and only if } \sigma(i) \models \psi, \text{ for all } \psi \in cl(\varphi) \quad (4.14)
\]

Since \( \sigma \models \varphi \), we have, by definition of global satisfaction that \( \sigma(0) \models \varphi \). Thus, \( \varphi \in \tau(0) \). It is obvious that \( \tau \) satisfies the valid closure labeling rules since they were built to mirror the semantics of LTL. \( \blacksquare \)

We now enunciate a Lemma that is a direct consequence of Lemmas 4.1 and 4.2 and gives a necessary and sufficient condition for the satisfaction of a formula \( \varphi \) by an \( \omega \)-string \( \sigma \), in terms of valid closure labelings.

**Lemma 4.3.** Let \( \varphi \in L_{\mathit{LTL}}(\Pi) \) and \( \sigma : \mathbb{N} \rightarrow 2^\Pi \). Then, \( \sigma \models \varphi \) if and only if there exists a valid closure labeling \( \tau : \mathbb{N} \rightarrow 2^{cl(\varphi)} \) of \( \sigma \) such that \( \varphi \in \tau(0) \).

Now, we are in conditions to present the construction of the Büchi Automaton \( B_\varphi \) and prove its correctness.
Theorem 4.1. Let $\varphi \in L_{\text{LTL}}(\Pi)$. Then the (generalized) Büchi automaton $B_\varphi = Ac(X,E,f,X_0,X_m)$ where

- $X \subseteq 2^{d(\varphi)}$ such that $x \in X$ if and only if
  - $\text{false} \notin x$
  - if $(\varphi_1 \land \varphi_2) \in x$ then $\varphi_1 \in x$ and $\varphi_2 \in x$
  - if $(\varphi_1 \lor \varphi_2) \in x$ then $\varphi_1 \in x$ or $\varphi_2 \in x$

(the states correspond to the sets of formulas in $\text{cl}(\varphi)$ that satisfy rules 1, 3 and 4 of Definition 4.6)

- $E = 2^\Pi$ (events correspond to sets of propositions)

- $f : X \times 2^\Pi \to 2^X$ such that $y \in f(x,e)$ if and only if
  - for all $p \in \Pi$, if $p \in x$ then $p \in e$
  - for all $p \in \Pi$, if $(\neg p) \in x$ then $p \notin e$
  - if $(X\varphi_1 \in x)$ then $\varphi_1 \in y$
  - if $(\varphi_1 U \varphi_2) \in x$ then either $(\varphi_2 \in x)$ or $(\varphi_1 \in x \land (\varphi_1 U \varphi_2) \in y)$
  - if $(\varphi_1 R \varphi_2) \in x$ then $\varphi_2 \in x$ and either $\varphi_1 \in x$ or $(\varphi_1 R \varphi_2) \in y$

- $X_0 = \{x \in X : \varphi \in x\}$

- Let $(\psi_1 U \psi'_1), \ldots, (\psi_m U \psi'_m)$ be all the formulas of the form $(\varphi_1 U \varphi_2)$ in $\text{cl}(\varphi)$. Then $X_m = \{X_1, \ldots, X_m\}$ where
  - $X_i = \{x \in X : (\psi_1 U \psi'_1) \land \psi'_i \in x \lor (\psi_1 U \psi'_2) \notin x\}$

is such that

$$\sigma \in L_m(B_\varphi) \text{ if and only if } \sigma \models \varphi \quad (4.15)$$

Proof. By Lemma 4.3, proving that $\sigma \in L_m(B_\varphi)$ if and only if exists a valid closure labeling $\tau : \mathbb{N} \to 2^{d(\varphi)}$ of $\sigma$ such that $\varphi \in \tau(0)$ suffices. By definition of marked language by a Büchi automaton, there exists a valid state labeling of $B$ and $\sigma$, $\rho : \mathbb{N} \to 2^{d(\varphi)}$ such that $\inf(\rho) \cap X \neq \emptyset$ for all $X \in X_m$. Take such $\rho$. It is easy to see that it is a valid closure labeling, i. e., that it respects all the rules of the valid closure labeling definition:

- By definition of $X$, rules 1, 3 and 4 are satisfied
- By definition of $f$ and by condition (2.20), rules 2, 5, 6 and 7 are satisfied
- By definition of $X_m$, rule 8 is satisfied:
  Given $(\varphi_1 U \varphi_2) \in \rho(i)$, $i \in \mathbb{N}$, we must guarantee the existence of $j \geq i$ such that $\varphi_2 \in \rho(j)$. We do this by imposing that there exists a state $x$ visited infinitely often such that
  - $(\varphi_1 U \varphi_2), \varphi_2 \in x$ meaning that there is always a state visited after $\rho(i)$ such that $(\varphi_1 U \varphi_2)$ has been propagated from $\rho(i)$ by rule 6 and $\varphi_2$ is also satisfied

35
or

\((- (\varphi_1 U \varphi_2)) \not\in x\) meaning that after some point \((\varphi_1 U \varphi_2)\) is no longer required to be satisfied.

Furthermore, by definition of \(X_0\) and by condition (2.19), \(\varphi \in \rho(0)\).

The translation from LTL to Büchi automata yields an automaton whose number of states is exponential on the size of the formula (the size of a formula is the number of operators the formula contains). The construction we just presented lacks efficiency and is not used in practice, since it is always exponential in the size of the formula. It is only useful to prove the existence of a Büchi automaton that marks the \(\omega\)-strings that satisfy a given LTL formula \(\varphi\), since its correctness is quite direct to establish. In Section 4.2, we will use a much more efficient implementation of the translation, which can be found in [4]. This implementation is also exponential in the worst case, but, through several simplifications, the size of the automaton is, in most cases, greatly reduced.
In this Chapter, we explain how to define the LTL-based supervisor for a plant \( G \) and a set of LTL formulas \( \varphi_1, \ldots, \varphi_n \), \( n \in \mathbb{N} \). In Section 5.1 we present the method of construction and in Section 5.2 we build supervisors for the transporting robots, dining philosophers and robotic soccer examples.

### 5.1 Defining the LTL-based Supervisor

As we have seen, the first step in building a standard realization of a supervisor \( S \), such that \( L(S/G) = L_\omega \) is to construct an FSA \( H_{\text{spec}} \) that captures the essence of our natural language specification. The construction of \( H_{\text{spec}} \) can be very error-prone and, in general, not obvious. But, as we have seen in Chapter 3, translating natural language to LTL formulas is, in most cases, straightforward. Thus, using the method presented in Section 4.2, our problem can be solved in a much more user-friendly way.

Note that, in order to restrict \( L(G) \) to \( L_\omega \), we will be constructing LTL formulas over the set of propositional symbols \( E \) (G’s event set), i.e., we will be interested in formulas \( \varphi \in L_{\text{LTLE}}(E) \). Since we assume the occurrence of events in a DES to be asynchronous, at each state exactly one event can occur. This allows us to assume \( \sigma : \mathbb{N} \rightarrow E \) in Definition 4.2 and substitute condition \( \sigma(t) \models p \) if and only if \( p \in \sigma(t) \) by \( \sigma(t) \models e \) if and only if \( \sigma(t) = e \), for \( t \in \mathbb{N} \) and \( e \in E \). Thus, given a Büchi automaton \( B_{\varphi} \), we can delete all events that are not singletons in \( B_{\varphi} \)’s event set and redefine \( B_{\varphi} \)’s transition function accordingly.

Since a Büchi automaton’s structure is the same as an NDFA, we consider \( B_{\varphi} \) as an NDFA. Next, we need to find the equivalent DFA, \( H_{\varphi} \), of \( B_{\varphi} \). This must be done because, if we build a supervisor from \( B_{\varphi} \), it will disable some events that should not be disabled, accordingly with the nondeterministic choices that are made when an event occurs at a given state and there is more than one state we can go to. This problem is solved by using the equivalent DFA, thus keeping track of all the states \( B_{\varphi} \) can be in and enabling all the events that are active in at least
one of those states. As we have seen, finding the equivalent DFA of an NDFA is an exponential operation, but, in general, the LTL formulas that are relevant to perform supervision yield small Büchi automata. Despite that, the complexity issue is a major one when applying this theory, as we will see in the next Section.

Then, we obtain the supervisor $S \varphi = G \parallel H \varphi$ or $S \varphi = G \times H \varphi$, depending on our supervision problem. Using this method, we guarantee that for all $s \in L(S \varphi/G)$, there exists $\sigma \in E^\omega$ such that $s \sigma \models \varphi$, i.e., the generated language of the CDES $S/G$ is always in conformity with the specification given by $\varphi$. Since the generated language by a CDES is a set of finite strings, this is the best we could hope for in this framework.

We can now describe the method we will use for supervision. Given a plant $G$ and a set of formulas $\{\varphi_1, ..., \varphi_n\}$, $n \in \mathbb{N}$ representing the specifications we want $G$ to fulfill, we build the supervisors $S_{\varphi_1}, ..., S_{\varphi_n}$, as explained above, and perform modular supervision, as explained in Section 3.3. The use of modular supervision gives us the gain in efficiency referred in Section 3.3 and, in addition, allows us to translate the formulas $\varphi_1, ..., \varphi_n$ to Büchi automata one by one, which also allows a great improvement in the efficiency of the method:

If $r_1, ..., r_n$ is the size (number of operators) of $\varphi_1, ..., \varphi_n$ respectively, then

- If we had not opted for modular control, to enforce all the specifications given by $\varphi_1, ..., \varphi_n$ we would need to build a Büchi automaton $B_\varphi$ for formula

  $$\varphi = \left( \bigwedge_{i=1}^{n} \varphi_i \right) \quad (5.1)$$

  It is easy to see that $\varphi$ has, at most, size

  $$r = \left( \sum_{i=1}^{n} r_i \right) + n - 1 \quad (5.2)$$

  where the $n - 1$ factor is due to the $n - 1$ "and" ($\land$) operators we added to $\varphi$. Hence, $B_\varphi$ would have, at most, the following number of states (we have seen that the translation from an LTL formula to a Büchi automaton yields an automaton whose number of states is exponential in the size of the formula):

  $$|B_\varphi| = 2^r \quad (5.3)$$

- Using modular supervision, we need to build $n$ Büchi automata $B_{\varphi_1}, ..., B_{\varphi_n}$, which, altogether, have at most the following total number of states:

  $$\sum_{i=1}^{n} |B_{\varphi_i}| = \sum_{i=1}^{n} 2^{r_i} \quad (5.4)$$

  which is clearly better than the previous option’s worst case scenario.
5.2 Examples

In this section, we present some applications of the framework defined throughout this work. We will build supervisors for the DES in Examples 3.3, 3.4 and 3.5 that enforce the specifications we gave in natural language in Examples 3.7, 3.8 and 3.9. To build these examples, some functions were implemented in Matlab. These function can be found in http://isabelle.math.ist.utl.pt/l52706/des:

- A function that receives a NFA and outputs its equivalent DFA.
- A function that receives two FSA and outputs their product composition.
- A function that receives two FSA and outputs their parallel composition.
- A function that receives a set of LTL formulas and translates them to Büchi automata (this function uses the implementation described in [4] to build the Büchi automaton, which is written in C and adapts a Matlab function written for the implementation described in [7] to take the output of the C function and turn it into a viable Matlab structure).
- A function that, given a plant and n supervisors, simulates the feedback loop of modular control.
- A function that congregates all of the above. This function receives a plant and n LTL formulas, creates the supervisors and simulates the feedback loop of modular control.

Example 5.1 (Transporting Robots). Let’s return to the transporting robots example and let G be the FSA represented in Example 3.3. In Example 3.7 we defined 4 specifications that prevent the robots from moving to a position where they are too far from the other, making the bar fall. Spec i can be translated to LTL by formula \( \varphi_i \), where

\[
\varphi_1 = (G(Move_2 \Rightarrow (X((\neg Move_1) U Stop_2)))) \\
\varphi_2 = (G(Move_1 \Rightarrow (X((\neg Move_2) U Stop_1)))) \\
\varphi_3 = (G(Move_1 \Rightarrow (X((\neg Move_1) U Stop_2)))) \\
\varphi_4 = (G(Move_2 \Rightarrow (X((\neg Move_2) U Stop_1))))
\]

Looking at these formulas, one can see that the events that can be disabled are Move_1 and Move_2. Hence, the controllability condition (3.9) is satisfied. We construct the DFA \( H_{\varphi_i} \), \( i = 1, 2, 3, 4 \) from the Büchi automata, as explained before. In Figure 5.1 we represent the Büchi automaton obtained from \( \varphi_2 \).

Next, we obtain the 4 supervisors \( S_i = G \parallel H_{\varphi_i} \). In Figure 5.2 we represent the supervisor \( S_2 \). Note that the states reached after an event Move_1 happens do not have the event Move_2 in their active event set.
Figure 5.1: B"uchi automaton marking the $\omega$-strings that satisfy $\varphi_2$

Figure 5.2: The supervisor $S_2$, obtained by formula $\varphi_2$

The modular supervisor $S_{mod1234}$ implements the robot alternation. The controlled system only allows 2 types of strings, $Move_1 - Stop_1 - Move_2 - Stop_2 - Move_1 - Stop_1 - Move_2 - Stop_2 - ...$ or $Move_2 - Stop_2 - Move_1 - Stop_1 - Move_2 - Stop_2 - Move_1 - Stop_1 - ...$. In Figure 5.3, we represent the automaton $G \times S_1 \times S_2 \times S_3 \times S_4$ which, as we have seen, represents the controlled system.

One should notice that our controlled system is not minimum, i.e., there is a 5 states DFA that implements the robot alternation. This is one drawback of this method: in general the controlled system is not the smallest it could be.

Example 5.2 (Dining Philosophers). As we have seen in Example 3.4, in the dining philosophers a deadlock may occur. As we explained in Example 3.8, one way to prevent the deadlock is by only allowing a philosopher to pick up the forks starting by the one with the minimum index. We can express this condition in LTL with the formulas $\varphi_{1,i}$ and $\varphi_{2,i}$, $i = 0, ..., n - 2$, where $\varphi_{2,i}$ says that, when philosopher $i$ stops eating, the next fork he will pick up will always be fork $i$ and $\varphi_{1,i}$ covers the case that is not taken into account in $\varphi_{2,i}$, saying that the first time philosopher $i$ picks up a fork, it will be fork $i$. 
Figure 5.3: Automaton representation of the controlled system, with the robot alternation implemented

\[ \varphi_{1,i} = (G_{i}f \Rightarrow (X((\neg_{i}f_{i+1})U_{i}f_{i}))) \]  
\[ \varphi_{2,i} = ((\neg_{i}f_{i+1})U_{i}f_{i}) \]

The case of philosopher \( n - 1 \) is expressed similarly, noticing that the forks near him are fork 0 and fork \( n - 1 \):

\[ \varphi_{1,n-1} = (G_{n-1}f \Rightarrow (X((\neg_{n-1}f_{n-1})U_{n-1}f_{0}))) \]

\[ \varphi_{2,n} = ((\neg_{n-1}f_{n-1})U_{n-1}f_{0}) \]

As we referred before, we consider the events \( if \) to be uncontrollable. Hence, we must guarantee that the controllability condition (3.9) is satisfied. Looking to the LTL formulas, one can easily see that the only events that will be disabled by the supervisors are \( if_{i+1}, \ i = 0, ..., n - 2 \) and \( n-1f_{n-1} \). Thus, it is immediate that the controllability condition holds.

The controlled system was tested for the case of 4 philosophers. The plant, resulting of the parallel composition of 4 automata and 4 forks has 81 states. The supervisors yield by \( \varphi_{1,i} \) and \( \varphi_{2,i} \) have 135 and 126 states respectively, for each \( i \). This is mostly due to the fact that, to obtain the supervisors, we must compose the specification automata \( H_{\varphi_{1,i}} \) and \( H_{\varphi_{2,i}} \) (both of them only have 3 states) with the plant, for each \( i \). Now, we give two examples of outputs of the simulation, with 100 events each. As expected, a deadlock never occurs and the philosophers always pick up the forks in order.

- 2f2 − 0f0 − 1f1 − 2f3 − 2f − 2f2 − 2f3 − 2f − 2f2 − 2f3 − 2f − 2f2 − 2f3 − 2f − 2f2 − 2f3 −
Regarding Example 3.9, it is easier to represent Spec 1, i.e.,

\[ \varphi \] allows more than one robot to have the ball in its possession and it is satisfied is immediate.

Example 5.3 (Robotic Soccer). Regarding Example 3.9, it is easier to represent Spec 1, i.e., all the robots will not move to the ball until one of them shoots it (which means that the team lost the ball possession).

Spec 2, i.e., all the robots will not move to the ball until one of them shoots it (which means that the team lost the ball possession).

These formulas do not refer to uncontrollable events, so checking that the controllability condition is satisfied is immediate.

The controlled system was tested for 3 robots. The plant has 729 states, the supervisor obtained by \( \varphi_1 \) has 100 states (the great reduction in the number of states is due to the fact that the plant allows more than one robot to have the ball in its possession and it is \( \varphi_1 \) that disallows this kind of situation) and the supervisors obtained by \( \varphi_{2,i} \), i.e., all the robots will not move to the ball until one of them shoots it (which means that the team lost the ball possession).

These formulas do not refer to uncontrollable events, so checking that the controllability condition is satisfied is immediate.

The controlled system was tested for 3 robots. The plant has 729 states, the supervisor obtained by \( \varphi_1 \) has 100 states (the great reduction in the number of states is due to the fact that the plant allows more than one robot to have the ball in its possession and it is \( \varphi_1 \) that disallows this kind of situation) and the supervisors obtained by \( \varphi_{2,i} \), i.e., all the robots will not move to the ball until one of them shoots it (which means that the team lost the ball possession).

These formulas do not refer to uncontrollable events, so checking that the controllability condition is satisfied is immediate.

Formula \( \varphi_1 \) enforces that, after one robot moves to the ball (which means the team does not have the ball in its possession), all the robots will not move to the ball until one of them shoots it (which means that the team lost the ball possession).

\[ \varphi_1 = (G[(\bigvee_i \text{move_to_ball}(i)) \Rightarrow (X[(\neg(\bigvee_i \text{move_to_ball}(i))) \cup (\bigvee_i \text{kick_ball}(i))])]) \] (5.13)

Formula \( \varphi_1 \) enforces that, after one robot moves to the ball (which means the team does not have the ball in its possession), all the robots will not move to the ball until one of them shoots it (which means that the team lost the ball possession).

\[ \varphi_{2,i} = ((\neg \text{start_receiving}(i)) \land (G[(\bigvee_{j \neq i} \text{start_passing}(j, i)) \Leftrightarrow (X \text{start_receiving}(i))])) \] (5.14)

Formula \( \varphi_{2,i} \) enforces that a robot’s first action cannot be getting ready to receive a pass and that, only when one of its teammates chooses it as a receiver, it gets ready to receive the ball and it gets ready as soon as possible.

These formulas do not refer to uncontrollable events, so checking that the controllability condition is satisfied is immediate.

The controlled system was tested for 3 robots. The plant has 729 states, the supervisor obtained by \( \varphi_1 \) has 100 states (the great reduction in the number of states is due to the fact that the plant allows more than one robot to have the ball in its possession and it is \( \varphi_1 \) that disallows this kind of situation) and the supervisors obtained by \( \varphi_{2,i} \), i.e., all the robots will not move to the ball until one of them shoots it (which means that the team lost the ball possession).

These formulas do not refer to uncontrollable events, so checking that the controllability condition is satisfied is immediate.

The controlled system was tested for 3 robots. The plant has 729 states, the supervisor obtained by \( \varphi_1 \) has 100 states (the great reduction in the number of states is due to the fact that the plant allows more than one robot to have the ball in its possession and it is \( \varphi_1 \) that disallows this kind of situation) and the supervisors obtained by \( \varphi_{2,i} \), i.e., all the robots will not move to the ball until one of them shoots it (which means that the team lost the ball possession).

These formulas do not refer to uncontrollable events, so checking that the controllability condition is satisfied is immediate.

The controlled system was tested for 3 robots. The plant has 729 states, the supervisor obtained by \( \varphi_1 \) has 100 states (the great reduction in the number of states is due to the fact that the plant allows more than one robot to have the ball in its possession and it is \( \varphi_1 \) that disallows this kind of situation) and the supervisors obtained by \( \varphi_{2,i} \), i.e., all the robots will not move to the ball until one of them shoots it (which means that the team lost the ball possession).

These formulas do not refer to uncontrollable events, so checking that the controllability condition is satisfied is immediate.
start_receiving(1) → pass(2.1) → move_to_goal(1) → blocked_path(1) → start_passing(1, 3) →
start_receiving(3) → pass(1, 3) → move_to_goal(3) → close_to_goal(3) → kick_ball(3) →
move_to_ball(3) → close_to_ball(3) → get_ball(3) → move_to_goal(3) → blocked_path(3) →
start_passing(3.1) → start_receiving(1) → pass(1, 2) → close_to_goal(2) → kick_ball(2) →
move_to_ball(3) → close_to_ball(3) → blocked_path(3) → start_passing(3, 2) → start_receiving(2) →
pass(3, 2) → move_to_goal(2) → kick_ball(2) → move_to_ball(2) → close_to_ball(2) → blocked_path(2) →
start_passing(2, 3) → start_receiving(3) → pass(2, 3) → move_to_goal(3) → blocked_path(3) →
start_passing(3, 1) → start_receiving(1) → pass(3, 1) → blocked_path(1) →
start_passing(1, 3) → start_receiving(3) → pass(1, 3) → close_to_goal(3) → kick_ball(3) →
move_to_ball(3) → close_to_ball(3) → get_ball(3) → move_to_goal(3) → blocked_path(3) →
start_passing(3, 2) → start_receiving(2) → pass(3, 2) → move_to_goal(2) → kick_ball(2) →
move_to_ball(2) → close_to_ball(2) → get_ball(2) → blocked_path(2) →
start_passing(2, 3) → start_receiving(3) → pass(2, 3) → close_to_goal(3) → kick_ball(3) →
move_to_ball(3) → close_to_ball(3) → get_ball(3) → close_to_goal(3) → kick_ball(3)

- move_to_ball(1) → close_to_ball(1) → get_ball(1) → blocked_path(1) → start_passing(1, 3) →
  start_receiving(3) → pass(1, 3) → move_to_goal(3) → blocked_path(3) → start_passing(3, 1) →
  start_receiving(1) → pass(3, 1) → blocked_path(1) → start_passing(1, 3) → start_receiving(3) →
  pass(1, 3) → move_to_goal(3) → close_to_goal(3) → kick_ball(3) → move_to_ball(2) →
  close_to_ball(2) → get_ball(2) → blocked_path(2) → start_passing(2, 3) → start_receiving(3) →
  pass(2, 3) → close_to_goal(2) → kick_ball(2) → move_to_ball(1) → close_to_ball(1) →
  get_ball(1) → close_to_goal(1) → kick_ball(1) → move_to_ball(2) → close_to_ball(2) →
  get_ball(2) → close_to_goal(2) → kick_ball(2) → move_to_ball(1) →

43
close_to_ball(1)−get_ball(1)−blocked_path(1)−start_passing(1,2)−start_receiving(2)−pass(1,2)−close_to_goal(2)−kick_ball(2)−move_to_ball(1)−close_to_ball(1)−get_ball(1)−move_to_goal(1)−close_to_ball(1)−pass(1,2)−close_to_ball(2)−get_ball(2)−move_to_goal(2)−blocked_path(2)−start_passing(2,1)−start_receiving(1)−pass(2,1)−close_to_goal(1)−kick_ball(1)−move_to_ball(1)−close_to_ball(1)−get_ball(1)−blocked_path(1)−start_passing(1,2)−start_receiving(2)−pass(1,2)−blocked_path(2)−start_passing(2,3)−start_receiving(3)−pass(2,3)−blocked_path(3)−start_passing(3,2)−start_receiving(2)−pass(3,2)−move_to_goal(2)−close_to_goal(2)−kick_ball(2)−move_to_ball(3)−close_to_ball(3)−get_ball(3)−blocked_path(3)−start_passing(3,1)−start_receiving(1)−pass(3,1)−move_to_goal(1)−close_to_goal(1)−kick_ball(1)−move_to_ball(1)−close_to_ball(1)−get_ball(1)−blocked_path(1)−start_passing(1,2)−start_receiving(2)−pass(1,2)−close_to_goal(2)−kick_ball(2)−move_to_ball(2)−close_to_ball(2)−get_ball(2)−blocked_path(2)−start_passing(2,3)−start_receiving(3)−pass(2,3)−blocked_path(3)
Conclusion

In this work, we defined a method to perform supervisory control of Discrete Event Systems using Linear-Time Temporal Logic. We introduced all the necessary theory to understand how the method works and gave some examples of application. Analyzing the examples, one can conclude that, with this method, the specification of supervisors for systems with an arbitrary number of components that must coordinate themselves is almost straightforward: all the formulas are written for an arbitrary $n \in \mathbb{N}$. Unfortunately, this advantage is somewhat shadowed by the high complexity of the method: despite writing the formulas for an arbitrary number of components, when performing the simulations we witnessed the great increase of the number of states, both in the plant and in the supervisors, which only allows the application of the method for systems with a relatively small number of components.

There are several paths one can follow to improve the method we just presented. The most obvious one is to try to improve its complexity. Another improvement is to increase the method’s expressive power, for example by using CTL (a temporal logic that is incomparable with LTL) or CTL$^*$ (a temporal logic that contains both LTL and CTL) ([6]) as a way to specify the supervisors or by identifying each state of the DES model with a set of propositions that are satisfied in that state and build our LTL specification over those propositions, instead of building it over the DES’ event set (one major advantage of this option is that it allows for more than one proposition to be satisfied at each state of the DES, unlike the method we presented, where only one is satisfied). One can also model the DES itself as a set of LTL formulas, as seen in [8], avoiding the construction of any automaton by hand (which can be very error-prone). Another option is to define a similar logic to LTL, but with its semantics defined over finite string, avoiding the need to use Büchi Automata. A final suggestion is to develop this theory in order to cover other aspects of Supervisory Control. For example, being concerned with marked languages and deal with blocking issues or introduce the notion of unobservable events ([2]).
Bibliography


[4] Paul Gastin and Denis Oddoux, Fast LTL to Büchi Automata Translation, LIAFA, Université Paris 7


47